MOTOROLA

APR9/D
Rev. 1

Full-Duplex 32-Kbit/s CCITT ADPCM Speech Coding

on the

Motorola DSP56001 Digital Signal Processors

Motorola's High-Performance DSP Technology dsp

Illustrations

Illustrations

List of Tables

need about four fewer bits per sample for equivalent speech quality. Therefore, only eight bits are needed per sample rather than the 11 or 12 bits that uniform PCM needs. Eight-bit log PCM companding is one of the simplest forms of speech coding and has also been standardized for digital telecommunications.

Speech coders are typically specified in terms of bit rate. The bit rate is the number of samples per second, times the number of bits per sample. As noted, telephone quality voice signals primarily range from 200 to 3200 Hz and they are typically sampled at 8 kHz to maintain this frequency range. Therefore, the basic standard data rate for μ-law and A-law PCM data is 8 bits/sample at 8000 samples/second or 64,000 bits/second (64 kbit/s). This is the source of the data rate for a basic digital transmission channel. The process of converting log PCM signals to/from analog signals is often done using CODEC devices (such as the MC145503). In the CCITT algorithm, the log PCM data is converted to linear PCM data before the ADPCM encoding itself is performed and the decoded linear signal is converted back into log PCM form after the decoding process is completed. The process for converting between log and linear PCM data, including the implementation on the DSP56001, is described in detail in the Motorola applications report “Logarithmic/Linear Conversion Routines For DSP56000/1” [6].

ADPCM techniques use a combination of APCM and DPCM techniques. There are many different ways to implement the concepts of APCM and DPCM. Therefore, the term ADPCM can justifiably refer to a broad range of speech coders that may have widely varying characteristics. ADPCM techniques, as well as APCM and DPCM techniques, may also be applied to non-speech signals, such as high-fidelity audio signals or video images. These implementations may not exploit properties that are specific to speech, however. The term ADPCM as used in this discussion refers specifically to the algorithm defined by the CCITT for telephone quality speech signals. Therefore the scope of this algorithm may not apply to all applications requiring signal compression. A general block diagram of the ADPCM configuration used in the CCITT algorithm is shown in Figure 3-7 [2]. ■

sixth order section that models zeroes and a second order section that models poles. The prediction filter shown in Figure 4-5, is implemented in the routines FMULT and ACCUM using the equations:

$$s_{ez}(k) = \sum_{i=1}^{6} b_i(k-1) \bullet d_q(k-i) \qquad \text{Eqn. 4-2}$$

$$s_e(k) = \sum_{i=1}^{2} a_i(k-1) \bullet s_r(k-i) + s_{ez}(k) \qquad \text{Eqn. 4-3}$$

The CCITT standard specifies that the multiplies in Eqn. 4-2 and Eqn. 4-3 be done in floating point so the values of $d_q(k)$ and $s_r(k)$ must be converted to floating point. This is done in the routines FLOATA and FLOATB. The signal $s_r(k)$ in Eqn. 4-2 and Eqn. 4-3 is the reconstructed signal. The routine ADDB calculates $s_r(k)$ by adding the quantized difference signal $d_q(k)$ to the signal estimate $s_e(k)$ as shown in Eqn. 4-4. The reconstructed signal represents the overall output of the ADPCM algorithm. The encoder does not output this signal but uses it as feedback for the prediction.

$$s_r(k-i) = s_e(k-i) + d_q(k-i) \qquad \text{Eqn. 4-4}$$

The predictor coefficients $a_i(k)$ and $b_i(k)$ are updated for each sample using a gradient search algorithm. The adaptation of the two pole coefficients, $a_1(k)$

The new difference signal $d_x(k)$ is then converted to the normalized logarithmic signal $d_{lnx}(k)$ in the routines LOG and SUBTB. The same quantization as in the encoder then occurs in the routine SYNC. But this routine also does a comparison of the new coded ADPCM signal to the received ADPCM signal I(k). The final PCM output of the decoder, $s_d(k)$, is determined by this comparison, defined by:

$$
\begin{aligned}
s_d(k) &= s_p(k)^+ && \text{if } d_x(k) < \text{lower interval boundary} \\
&= s_p(k)^- && \text{if } d_x(k) \geq \text{upper interval boundary} \\
&= s_p(k) && \text{otherwise}
\end{aligned}
$$

Eqn. 4-25

where:

$s_p(k)+$ = the PCM code word that represents the next more positive PCM output level

$s_p(k)-$ = the PCM code word that represents the next more negative PCM output level

■

ADPCMNS.ASM is the program for the non-standard version and it is a modification of the standard implementation. ADPCMNS.ASM performs the complete ADPCM algorithm in real-time on a single DSP56001, and requires less computational power than the standard version. In addition to providing a more efficient implementation of the ADPCM algorithm, this version is better suited for modification since the algorithm is programmed more directly than the CCITT standard specification allows.

For both ADPCM versions, the encoder and decoder portions of the source code are designed to be independent of the I/O interfaces so that the code can be easily modified for a variety of configurations, including single or multiple channel half-duplex configurations. For the standard version, this feature permits real-time performance on a slower speed DSP56001 or allows other simultaneous tasks to be performed on the same 27 MHz DSP56001. For the non-standard version, this feature allows an even greater variety of configurations. Further performance details for both versions are described in ***SECTION 5.5 Performance Specifications.*** This section details the implementation of the CCITT ADPCM algorithm on the DSP56001.

This application report provides only a basic description of the source code. For a more complete understanding of the DSP56001 code, refer to the CCITT document. Many of the details in Recommendation G.721 are not included in this document but have a significant impact on the assembly implementations, especially the standard version. In

also discussed in the ADPCM.HLP file. The data in the two files being processed does not have to be related in any way since the encoder and decoder are designed to operate on two independent signals. Additionally, any file containing PCM data in ASCII hex characters may be used as input to the encoder, and likewise any file containing ADPCM data can be used as input to the decoder. It should be noted however that the file I/O routines on the ADS are not designed for high-speed data transfer so that processing data files with the DSP56001 ADPCM program will not be in real-time.

The non-standard ADPCM program includes the code required for a real-time I/O interface. The PCM channel is provided by a Motorola MC145503 CODEC connected to the DSP56001's Synchronous Serial Interface (SSI). Eight general-purpose I/O pins on the DSP56001 are used for the ADPCM channel. Four are used for parallel input and the other four are used for parallel output.

The SSI interface (both the transmit and receive) and the parallel I/O interface are assumed to be synchronous. The code is synchronized with the I/O interface by polling. No interrupts are used, although they can be added if desired. The real-time full-duplex operation of the non-standard ADPCM program has been tested on a set-up consisting of two DSP56000ADS. Further details of this test set-up can be found in the file ADPCMNS.HLP.

As mentioned previously, the I/O interface of the ADPCM programs are designed to be flexible for a variety of configurations. The standard ADPCM

the various portions of the algorithm are executed. The order of execution of the individual CCITT routines along with their execution speed is given in ***SECTION 5.4 Performance Specifications***.

The encoder and decoder routines are designed to operate as independent code segments. They do assume that appropriate variables are stored in data memory and that appropriate pointers have been set. In particular, address registers r1, r2, r6, and r7 should contain the appropriate memory addresses prior to executing the encoder and decoder sections. Registers r3 and r5 should contain constant address values used for table lookup. Registers r0 and r4 do not need to be initialized since they are used as general purpose registers. The encoder and decoder are not set up as subroutines in the program. If interrupts are used for data I/O, they can easily be made into subroutines or interrupt routines. However, one routine should not interrupt the other routine until it is completely finished executing.

No subroutines are used within the encoder or the decoder so that optimal speed can be obtained. The code has also been optimized to take advantage of the DSP56001's architecture as much as possible. This causes the various CCITT routines in the code to “overlap” in many cases, meaning that variables and data values for one routine may be read from memory while the previous routine is still executing. In one case, the XOR and UPB routines are actually combined into a single section of code.

needed for the log PCM conversion routine. The zero wait states for external program memory are needed so the algorithm will run in real time.

The program initialization is done in the subroutine INIT. First, all internal X and Y data RAM is cleared, all variables that require specific values are initialized, and then all lookup tables are copied from their load-time locations in program memory to their run-time locations in data RAM. Next, the pointers to the receive (decode) and transmit (encode) data buffers are initialized. These buffers hold the delayed values of $d_q(k)$ and $s_r(k)$ used in the linear predictor filter. These are the only true modulo buffers used in the assembly code in the sense that the newest delayed values replace the oldest delayed values without actually moving the other delayed values. The INIT routine initializes the sign and mantissa locations in these buffers since the code assumes a certain range of legal values in these locations. The INIT routine also initializes other variables including the variable LAW. It determines whether the μ-law or A-law format is chosen. The program defaults to setting LAW to zero to select μ-law for the PCM format. The code can be changed to set LAW to any non-zero value which will select the A-law format. It can also be easily modified to select μ-law or A-law based on an external input.

The remaining portions of the INIT routine initialize the addressing registers and modifier registers that are used to access the data memory. Six of the variable storage areas (indicated by * in Figure 5-2) are addressed using modulo pointers. These six areas, three each for the encoder and decoder, are used by the linear predictor filter.

8-bit log PCM sample to a linear 14-bit two's-complement representation that is suitable for numeric operations. COMPRESS performs the opposite conversion. The DSP56001 ADPCM implementation supports both μ-law and A-law format conversion.

The routine EXPAND performs the conversion by using the internal μ-law and A-law ROM tables on the DSP56001. EXPAND uses register r3 as the pointer into the lookup table. Register r3 is set during program initialization to either the μ-law table base or the A-law table base and remains set to this value while the program is running. Register n3 is used as an offset into the ROM table. The assembly code for EXPAND is identical for both formats and is only dependent on the base pointer stored in r3 so separate routines are not needed. To obtain optimal speed, the COMPRESS routine requires separate code sections for the μ-law and A-law conversion. In this routine, the variable LAW is tested for each sample. If only one format is used, the variable LAW may be eliminated and one section of COMPRESS may be removed to save program memory. The SYNC routine discussed in ***SECTION 5.2.11*** also requires separate code segments for each log PCM format, so similar program memory savings can be obtained there.

The EXPAND and COMPRESS routines are modified versions of routines given in the Motorola applications brief “Logarithmic/Linear Conversion Routines for the DSP56000/1” [6]. They were chosen based on maximum execution speed. Complete descriptions of each routine can be found in the above applications brief.

each iteration until a 0 is in bit 23 and a 1 is in bit 22 (this is the normalized fraction format on the DSP56001). For each left shift the value in r0 is decremented. Once the magnitude has been normalized successive iterations will do no further adjustments. Fourteen iterations are performed since the maximum that the magnitude can be shifted is 14 bits, assuming the magnitude is non-zero. Fourteen iterations of the NORM is not needed in all cases but taking time to test after each iteration would cause the worst case delay to be longer. After the normalization process is finished the normalized mantissa will be in accumulator a and the associated exponent will be in register r0.

The remaining instructions combine the exponent and mantissa into a mixed number. The truncation of the mantissa to seven bits is performed by using a mask instead of actually shifting. This technique is common throughout the code. The process of combining the exponent and mantissa also shows the technique of shifting by multiplication. The exponent is moved from r0 to x1 where it will be in the four LSBs but it needs to be left justified to bits 22-19 which are the four MSBs of the DSP56001's fractional format. A shift constant is read from the shift constant table in Y memory and is multiplied with the exponent. The result is that the exponent is effectively shifted left 19 bits. A shift constant is also used to shift the mantissa right by three bits. In this example the mantissa is shifted right and combined with the exponent in a single MAC instruction. This shift technique is described in further detail in ***SECTION 5.4***. The resulting log signal DL is a mixed number with

```
;        QUAN
;   Quantize difference signal in log domain
;
;   log2 |D(k)| - Y(k)   | |I(k)|
;   -------------------- |-------
;        [3.12, + inf)   |   7
;        [2.72, 3.12)    |   6
;        [2.34, 2.72)    |   5
;        [1.91, 2.34)    |   4
;        [1.38, 1.91)    |   3
;        [0.62, 1.38)    |   2
;        [-0.98, 0.62)   |   1
;        (-inf, -0.98)   |   0
; Inputs:
;   DLN = siii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
;   DS = sXXX XXXX | XXXX XXXX | 0000 0000 (lTC) in reg Yl
; Output:
;    I = siii 0000 | 0000 0000 | 0000 0000 (ADPCM format) in accum A
;*************************************************************************
;   Quantization table in X memory
; QUANTAB     DC   $F89000            ;-0.98
;             DC   $050000            ;0.62
;             DC   $0B2000            ;1.38
;             DC   $0F6000            ;1.91
;             DC   $12C000            ;2.34
;             DC   $15D000            ;2.72
;             DC   $190000            ;3.12
;             DC   $7FFFFF            ;15.99
;

          MOVE     #QUANTAB,R0        ;Get quantization table base
          MOVE     #>QUANTAB+2,X1     ;Get offset for quan. conversion
          MOVE     X:(R0)+,X0         ;Get 1st quan. table value
TSTDLN_T
          CMP      X0,A   X:(R0)+,X0  ;Compare to DLN, get next value
          JGE      <TSTDLN_T          ;If value<DLN try next range
          MOVE     R0,A
          SUB      Xl,A   Y:LSHFT-20,X0   ;When range found subtract pointer
                                      ; from base to get IMAG=II

; A1 = 0000 0000 | 0000 0000 | 0000 0iii (A2=A0=0)
          MOVE     Al,X1
          MPY      X0,Xl,A     Yl,B   ;Shift IMAG <<20, result is
                                      ; in A0, move DS into B
          MOVE     A0,A
; A1 = 0iii 0000 | 0000 0000 | 0000 0000 (A2=A0=0)
          MOVE     Al,X:IMAG          ;Save IMAG
          TST      A       #<$F0,X0   ;Check IMAG, get invert mask
          JEQ      <INVERT_T          ;If IMAG=0 invert bits
          TST      B                  ; else check DS
          JPL      <IOUT_T            ;If DS=1 don't invert IMAG
INVERT_T  EOR      X0,A               ;If DS=0 or IMAG=0 invert IMAG
IOUT_T    MOVE     Al,A               ;Adjust sign extension
```

Figure 5-6 Difference Signal Scaling and Quantization (sheet 2 of 2)

The data buffer structure also allows the efficient offset addressing modes of the DSP56001 to be used when consecutive values of one component are required. For instance, the signs of each delayed $d_q(k)$ value are needed in the XOR/UPB routine. In this routine the offset register n3 is set to 3 so that when one sign is read from the buffer, register r3 is automatically post-incremented by 3 to point to the next sign. The modulo pointer feature of the DSP56001 also reduces execution speed since less software overhead is required to update the pointer. The data buffer for the prediction filter is actually a modified form of the usual modulo buffer structure on the DSP56001 since two new values, $s_r(k-1)$ and $d_q(k-1)$, are added to the buffer for each sample. The new $s_r(k-1)$ overwrites the current $d_q(k-6)$ and the new $d_q(k-1)$ overwrites the current $s_r(k-2)$. The coefficient buffer and partial product buffer are addressed using modulo pointers for efficiency but the physical locations of each component do not change.

The implementation of the ACCUM routine that adds the partial products is more straightforward than the FMULT routine. The $wb_i(k)$ partial products for the six zeros are accumulated first to obtain the partial signal estimate $s_{ez}(k)$. Then the $wa_i(k)$ partial products for the two poles are accumulated with the zeros to form the final signal estimate $s_e(k)$. Even though FMULT and ACCUM account for a large percentage of the overall execution speed they only implement Eqn. 4-2 and Eqn. 4-3 of the CCITT algorithm.

culates Eqn. 4-6 to update $a_2(k)$ in a similar manner. This routine is more complicated since it also has to calculate the value of $f(a_1)$ in Eqn. 4-8. After updating $a_1(k)$ and $a_2(k)$, the routines LIMD and LIMC limit these values. The routine LIMC uses a constant upper and lower limit defined in Eqn. 4-9 for the comparison. It makes use of the conditional transfer instructions of the DSP56001 as discussed in ***SECTION 5.4 Optimization Techniques***. The LIMD routine is very similar but it must calculate the upper and lower limits before the comparison.

The $b_i(k)$ coefficients are updated in the routines XOR and UPB which are combined in a single code segment. The XOR routine accounts for the sgn[x] multiplication in Eqn. 4-11 that uses the delayed values of $d_q(k)$. The calculation of Eqn. 4-11 is similar to that of Eqn. 4-5 and Eqn. 4-6 except that the calculation must be done six times for each of the six $b_i(k)$ coefficients. In this routine the special case of $sgn[d_q(k)] = 0$ is checked only once since it applies to all six calculations. If this case is found a DO loop that does not add the second half of Eqn. 4-11 is executed. If this case is not present a DO loop that does the full calculation is executed, but execution time is saved since the test for the special case does not have to be performed for each stage of the loop.

After the coefficients have been updated they are passed through the predictor trigger routine. Since this routine also affects the tone detect signal $t_r(k)$, they are all updated at once in the tone detection section of the algorithm. The final step of the adaptive prediction section is the conversion of the

as the standard version but eliminates many of the details described in Recommendation G.721. Two key features have been removed; one being the use of floating-point multiplies in the adaptive prediction filter. Instead, the non-standard code implements the filter in fixed-point arithmetic using the 24-bit multiplier and 56-bit accumulator on the DSP56001. The removal of this feature alone greatly improves the execution speed of the code and also reduces the code complexity. Another key feature of the standard ADPCM code that was removed is the synchronous coding adjustment block in the decoder. As noted previously, this block is included to prevent cumulative distortions when multiple ADPCM encodes and decodes are performed on the same channel. Since many applications do not require this block it has been removed in the non-standard version. Other minor details have also been removed to make individual routines as efficient as possible.

In addition to improving the execution speed of the algorithm, the non-standard version also reduces memory size requirements. The standard ADPCM code is not able to implement code segments as subroutines because of execution speed requirements. Since the execution speed of the non-standard routine is much faster, code segments that are common to the encoder and the decoder can be shared. This feature allows the run-time portion of the ADPCM algorithm to fit into the DSP56001's on-board program RAM, so that no high-speed external RAM is required for real-time operation of a single full-duplex channel. Even with

Figure 5-9 shows the memory map of the internal data RAM. In general, the arrangement of the variables and lookup tables is similar to the standard version. A notable difference between the two is the total storage requirement. A reason for this is that the predictor data buffers no longer require data to be stored in floating point format, so each buffer is only eight words long instead of 24. Also, since the predictor filter is implemented as one routine, the partial products of each multiply no longer need to be stored. Another reduction results from the elimination of the FMULT constant storage and the miscellaneous constant storage. Most of these constants are no longer needed since many are related to details of the CCITT specification. To reduce code complexity the remaining miscellaneous constants are stored as immediate data in program memory. Another difference from the standard version is the addition of a lookup table for the F[l] variable calculated in the routine FUNCTF.

Figure 5-10 shows how the non-standard version uses addressing registers. Registers r2, r3, and r6 function the same as in the standard code. The predictor data buffer is now addressed as a modulo 7 block instead of a modulo 23 block since the floating point format is no longer used. The ADPCM code no longer uses registers r1, r4, r5, and r7. Only register r0 is retained as a general purpose register. The use of register r3 can be removed if desired since the algorithm uses it only in the EXPAND routine. Register r0 can be used in this routine instead but it must be set to the correct PCM table base, based on the variable LAW, each time the CONVERT subroutine is called.

struction followed by a shift instruction. As noted previously, this process is iterative and requires overhead for the DO or REP. A faster way calls for multiplying the operand by a shift constant. If the amount of shift is always the same, the constant can be explicitly coded in the instruction sequence. If however, the shift amount is not always the same there is a convenient method of getting the appropriate shift constant. This method uses a lookup table for determining the shift constant. The table should be of the following form:

```
              org     x:

rshift        equ     *-1
              dc      $400000         ;>>1 or <<23
              dc      $200000         ;>>2 or <<22
              dc      $100000         ;>>3 or <<21
                      •••
              dc      $000002         ;>>22 or <<2
              dc      $00000l         ;>>23 or <<1
lshift        equ     *
```

This table can be put in either X or Y memory. To perform an arbitrary right shift the value of rshift should be loaded into one of the address registers as a table base. The amount of the shift will then be loaded into the corresponding offset register so that the appropriate shift constant can be read from the table in memory. The technique is similar for an arbitrary left shift although the offset will be negative. For either a left or right shift the integer shift amount should be between 1 and 23. The following is an example of a right shift:

```
move      #rshift,r0          ;Set r0=table base
move      b0,n0               ;Load shift amount as offset
move      al,xl               ;Set data up for shift
move      x:(r0+n0),x0        ;Lookup shift constant
mpy       x0,xl,a             ;Shift data right
                              ;Result is shifted into a0
```

In this example, the 24-bit number to be shifted is in a1 and the amount of the shift is in the LSBs of b0. The base of the right shift lookup table rshift is loaded into r0 and the amount of the shift is loaded into n0 as an offset into the table. The shift constant is found by using the Indexed by Offset addressing mode. The table base in r0 is not changed so the r0 does not have to be loaded again for another shift unless r0 is used elsewhere. To accomplish the shift the shift constant is put in x0 and the 24-bit number that is to be shifted is put in x1. The MPY instruction performs the shift. The result is found in a1 and a0. It is also sign extended into a2. This method is faster than the REP or DO method because the actual shift takes only one instruction cycle. Data movement is the only overhead and it can often be done in parallel with other operations. The only other extra time needed is the one extra instruction cycle required by the Indexed by Offset mode.

The following is an example of a left shift:

```
move     #1shift,r0          ;Set r0=table base
neg      b         a1,x1     ;Find negative shift amount,
                             ; set data up for shift
move     b0,n0               ;Load s.a. as negative offset
move     x:(r0+n0),x0        ;Lookup shift constant
mpy      x0,xl,a             ;Shift data left
                             ;Result is shifted into al
```

This example is very similar to the right shift except that a negative offset is used. The shift amount is negated before it is loaded as an offset, since the Indexed by Offset mode can only use a positive offset. Again the result is found in a0 and a1 and is

sign extended into a2. The left shift requires the same amount of data movement as the right shift but also needs an extra ALU operation to negate the shift amount. In the example shown, the execution time is the same as the right shift.

The APDCM code uses this technique in several forms. The shift table is in Y memory and uses the labels RSHFT and LSHFT. In some cases the table is addressed like the above examples but with register r5 as the pointer. When the immediate short addressing mode can be used, the table is addressed using this mode instead of the address pointer. In addition, some shift constants are addressed as immediate operands in the instruction word. In all cases the actual shift is performed in the same manner as the above examples.

As mentioned, many of these optimization techniques improve execution speed at the expense of memory usage or code complexity. Since execution speed is not as high a priority in the non-standard version, several of these techniques are not used. The result is greater memory savings and lesser code complexity. Some techniques, such as shifting by multiplication, are still used in some cases however. As with almost any program there are always many trade-offs to be considered. If a particular application requires greater speed from the non-standard version, the optimization techniques can be added to the code following the examples shown in the standard version.

No samples of the CCITT test sequences were observed to exceed the real-time limit and no indications of this occurrence were found in the real-time test set-up. If an extra margin is desired, the input clock can be increased to 27.5 MHz. Also the synchronization block in the decoder is not necessary for the ADPCM algorithm itself. This includes the routines EXPAND, SUBTA, LOG, SUBTB, and SYNC. This block is included for synchronization of multiple PCM/ADPCM/PCM conversions on a single channel. If only one PCM/ADPCM/PCM conversion is used the deletion of this block should not significantly affect the output speech quality. In this case the worst case execution time will be 1589 instruction cycles. Note that these routines are necessary to correctly pass the CCITT test sequences.

The non-standard version takes a total of 984 instruction cycles for both the encode and decode, indicating full-duplex operation is possible on a 20.5 MHz DSP56001. As noted, the non-standard implementation removed some of the optimizations used in the standard version to conserve program memory. If desired, these optimizations can be added to the non-standard version to improve performance. At least 100 instruction cycles can be saved from both the encoder and the decoder. This would allow 2 full-duplex channels on a 27 MHz DSP56001.

This application note uses the following terminology:

. = location of implied radix point

i = integer bit

f = fraction bit

s = sign bit

m = mantissa bit

e = exponent bit

1 = bit is always 1

0 = bit is always 0

X = bit value is unknown but is not significant

An ***exception*** to the above list is the PCM word where:

p = sign bit

s = segment bit

q = quantization level bit

Note that when labels are used to refer to variables or program locations in the assembly code the suffix "_T" refers to those labels associated with the encoder (transmit) and the suffix "_R" refers to those associated with the decoder (receive).

Example:

```
; y = 0iiii i.fff | ffff ff00 | 0000 0000 (13sm)
Y_T                   DS             1              ;Quantizer scale factor
```

The above example shows the memory allocation for a variable — the scale factor y(k). It is defined as a 13-bit signed magnitude number with four integer bits and nine fractional bits. The value stored in memory is always stored in the 24-bit format with the implied radix point between bits 18 and 19.

Example:

```
; A1 = 01mm mmm0 | 0000 0000 | 0000 0000 (A2=A0=0)
; B1 = 0000 0000 | 0000 0000 | 0000 eeee (B2=B0=0)
```

At the point where these comments appear in the code; the register a1 always contains a 1 in bit 22, five other mantissa bits, and all other bits set to 0. Register b1 always contains four exponent bits in bits 0 through 3 with all other bits set to 0. Registers a0, a2, b0, and b2 are always set to 0. ■